2.2 — Function return values (value-returning functions)

ALEX[1] FEBRUARY 6, 2024

Consider the following program:

```
#include <iostream>

int main()
{
    // get a value from the user
    std::cout << "Enter an integer: ";
    int num{};
    std::cin >> num;

    // print the value doubled
    std::cout << num << " doubled is: " << num * 2 << '\n';

    return 0;
}
```

This program is composed of two conceptual parts: First, we get a value from the user. Then we tell the user what double that value is.

Although this program is trivial enough that we don't need to break it into multiple functions, what if we wanted to? Getting an integer value from the user is a well-defined job that we want our program to do, so it would make a good candidate for a function.

So let's write a program to do this:

```
// This program doesn't work
#include <iostream>

void getValueFromUser()
{
    std::cout << "Enter an integer: ";
    int input{};
    std::cin >> input;
}

int main()
{
    getValueFromUser(); // Ask user for input

    int num{}; // How do we get the value from getValueFromUser() and use it
to initialize this variable?

    std::cout << num << " doubled is: " << num * 2 << '\n';

    return 0;
}
```

While this program is a good attempt at a solution, it doesn't quite work.

When function `getValueFromUser` is called, the user is asked to enter an integer as expected. But the value they enter is lost when `getValueFromUser` terminates and control returns to `main`. Variable `num` never gets initialized with the value the user entered, and so the program always prints the answer `0`.

What we're missing is some way for `getValueFromUser` to return the value the user entered back to `main` so that `main` can make use of that data.

Return values

When you write a user-defined function, you get to determine whether your function will return a value back to the caller or not. To return a value back to the caller, two things are needed.

First, your function has to indicate what type of value will be returned. This is done by setting the function's **return type**, which is the type that is defined before the function's name. In the example above, function `getValueFromUser` has a return type of `void` (meaning no value will be returned to the caller), and function `main` has a return type of `int` (meaning a value of type `int` will be returned to the caller). Note that this doesn't determine what specific value is returned -- it only determines what *type* of value will be returned.

Related content

We explore functions that return `void` further in the next lesson (2.3 -- Void functions (non-value returning functions) (https://www.learncpp.com/cpp-tutorial/void-functions-non-value-returning-functions/)[2]).

Second, inside the function that will return a value, we use a **return statement** to indicate the specific value being returned to the caller. The specific value returned from a function is called

the **return value**. When the return statement is executed, the function exits immediately, and the return value is copied from the function back to the caller. This process is called **return by value**.

Let's take a look at a simple function that returns an integer value, and a sample program that calls it:

```
#include <iostream>

// int is the return type
// A return type of int means the function will return some integer value to the caller (the specific value is not specified here)
int returnFive()
{
    // the return statement indicates the specific value that will be returned
    return 5; // return the specific value 5 back to the caller
}

int main()
{
    std::cout << returnFive() << '\n'; // prints 5
    std::cout << returnFive() + 2 << '\n'; // prints 7

    returnFive(); // okay: the value 5 is returned, but is ignored since main() doesn't do anything with it

    return 0;
}
```

When run, this program prints:

```
5
7
```

Execution starts at the top of `main`. In the first statement, the function call to `returnFive` is evaluated, which results in function `returnFive` being called. Function `returnFive` returns the specific value of `5` back to the caller, which is then printed to the console via `std::cout`.

In the second function call, the function call to `returnFive` is evaluated, which results in function `returnFive` being called again. Function `returnFive` returns the value of `5` back to the caller. The expression `5 + 2` is evaluated to produce the result `7`, which is then printed to the console via `std::cout`.

In the third statement, function `returnFive` is called again, resulting in the value `5` being returned back to the caller. However, function `main` does nothing with the return value, so nothing further happens (the return value is ignored).

Note: Return values will not be printed unless the caller sends them to the console via `std::cout`. In the last case above, the return value is not sent to `std::cout`, so nothing is printed.

Tip

When a called function returns a value, the caller may decide to use that value in an expression or statement (e.g. by using it to initialize a variable, or sending it to `std::cout`) or ignore it (by doing nothing else). If the caller ignores the return value, it is discarded (nothing is done with it).

Fixing our challenge program

With this in mind, we can fix the program we presented at the top of the lesson:

```
#include <iostream>

int getValueFromUser() // this function now returns an integer value
{
    std::cout << "Enter an integer: ";
    int input{};
    std::cin >> input;

    return input; // return the value the user entered back to the caller
}

int main()
{
    int num { getValueFromUser() }; // initialize num with the return value
of getValueFromUser()

    std::cout << num << " doubled is: " << num * 2 << '\n';

    return 0;
}
```

When this program executes, the first statement in `main` will create an `int` variable named `num`. When the program goes to initialize `num`, it will see that there is a function call to `getValueFromUser()`, so it will go execute that function. Function `getValueFromUser`, asks the user to enter a value, and then it returns that value back to the caller (`main`). This return value is used as the initialization value for variable `num`.

Compile this program yourself and run it a few times to prove to yourself that it works.

Revisiting main()

You now have the conceptual tools to understand how the `main` function actually works. When the program is executed, the operating system makes a function call to `main`. Execution then jumps to the top of `main`. The statements in `main` are executed sequentially. Finally, `main` returns an integer value (usually `0`), and your program terminates. The return value from `main` is sometimes called a **status code** (also sometimes called an **exit code**, or rarely a **return code**), as it is used to indicate whether the program ran successfully or not.

By definition, a status code of `0` means the program executed successfully.

Best practice

Your `main` function should return the value `0` if the program ran normally.

A non-zero status code is often used to indicate failure (and while this works fine on most operating systems, strictly speaking, it's not guaranteed to be portable).

For advanced readers

The C++ standard only defines the meaning of 3 status codes: 0, EXIT_SUCCESS, and EXIT_FAILURE. 0 and EXIT_SUCCESS both mean the program executed successfully. EXIT_FAILURE means the program did not execute successfully.

EXIT_SUCCESS and EXIT_FAILURE are preprocessor macros defined in the <cstdlib> header:

```
#include <cstdlib> // for EXIT_SUCCESS and EXIT_FAILURE

int main()
{
    return EXIT_SUCCESS;
}
```

If you want to maximize portability, you should only use 0 or EXIT_SUCCESS to indicate a successful termination, or EXIT_FAILURE to indicate an unsuccessful termination.

We cover the preprocessor and preprocessor macros in lesson 2.10 -- Introduction to the preprocessor (https://www.learncpp.com/cpp-tutorial/introduction-to-the-preprocessor/)[3].

C++ disallows calling the `main()` function explicitly.

As an aside...

C does allow `main()` to be called explicitly, so some C++ compilers will allow this for compatibility reasons.

For now, you should also define your `main()` function at the bottom of your code file, below other functions, and avoid calling it explicitly.

A value-returning function that does not return a value will produce undefined behavior

A function that returns a value is called a **value-returning function**. A function is value-returning if the return type is anything other than `void`.

A value-returning function *must* return a value of that type (using a return statement), otherwise undefined behavior will result.

Related content

We discuss undefined behavior in lesson 1.6 -- Uninitialized variables and undefined behavior (https://www.learncpp.com/cpp-tutorial/uninitialized-variables-and-undefined-behavior/)[4].

Here's an example of a function that produces undefined behavior:

```
#include <iostream>

int getValueFromUserUB() // this function returns an integer value
{
    std::cout << "Enter an integer: ";
    int input{};
    std::cin >> input;

    // note: no return statement
}

int main()
{
    int num { getValueFromUserUB() }; // initialize num with the return
value of getValueFromUserUB()

    std::cout << num << " doubled is: " << num * 2 << '\n';

    return 0;
}
```

A modern compiler should generate a warning because `getValueFromUserUB` is defined as returning an `int` but no return statement is provided. Running such a program would produce undefined behavior, because `getValueFromUserUB()` is a value-returning function that does not return a value.

In most cases, compilers will detect if you've forgotten to return a value. However, in some complicated cases, the compiler may not be able to properly determine whether your function returns a value or not in all cases, so you should not rely on this.

Best practice

Make sure your functions with non-void return types return a value in all cases.

Failure to return a value from a value-returning function will cause undefined behavior.

Function main will implicitly return 0 if no return statement is provided

The only exception to the rule that a value-returning function must return a value via a return statement is for function `main()`. The function `main()` will implicitly return the value `0` if no return statement is provided. That said, it is best practice to explicitly return a value from `main`, both to show your intent, and for consistency with other functions (which will exhibit undefined behavior if a return value is not specified).

Functions can only return a single value

A value-returning function can only return a single value back to the caller each time it is called.

Note that the value provided in a return statement doesn't need to be literal -- it can be the result of any valid expression, including a variable or even a call to another function that returns a value. In the `getValueFromUser()` example above, we returned a variable `input`, which held the number the user input.

There are various ways to work around the limitation of functions only being able to return a single value, which we'll cover in future lessons.

The function author can decide what the return value means

The meaning of the value returned by a function is determined by the function's author. Some functions use return values as status codes, to indicate whether they succeeded or failed. Other functions return a calculated or selected value. Other functions return nothing (we'll see examples of these in the next lesson).

Because of the wide variety of possibilities here, it's a good idea to document your function with a comment indicating what the return values mean. For example:

```
// Function asks user to enter a value
// Return value is the integer entered by the user from the keyboard
int getValueFromUser()
{
    std::cout << "Enter an integer: ";
    int input{};
    std::cin >> input;

    return input; // return the value the user entered back to the caller
}
```

Reusing functions

Now we can illustrate a good case for function reuse. Consider the following program:

```
#include <iostream>

int main()
{
    int x{};
    std::cout << "Enter an integer: ";
    std::cin >> x;

    int y{};
    std::cout << "Enter an integer: ";
    std::cin >> y;

    std::cout << x << " + " << y << " = " << x + y << '\n';

    return 0;
}
```

While this program works, it's a little redundant. In fact, this program violates one of the central tenets of good programming: **Don't Repeat Yourself** (often abbreviated **DRY**).

Why is repeated code bad? If we wanted to change the text "Enter an integer:" to something else, we'd have to update it in two locations. And what if we wanted to initialize 10 variables instead of 2? That would be a lot of redundant code (making our programs longer and harder to understand), and a lot of room for typos to creep in.

Let's update this program to use our `getValueFromUser` function that we developed above:

```
#include <iostream>

int getValueFromUser()
{
    std::cout << "Enter an integer: ";
    int input{};
    std::cin >> input;

    return input;
}

int main()
{
    int x{ getValueFromUser() }; // first call to getValueFromUser
    int y{ getValueFromUser() }; // second call to getValueFromUser

    std::cout << x << " + " << y << " = " << x + y << '\n';

    return 0;
}
```

This program produces the following output:

```
Enter an integer: 5
Enter an integer: 7
5 + 7 = 12
```

In this program, we call `getValueFromUser` twice, once to initialize variable `x`, and once to initialize variable `y`. That saves us from duplicating the code to get user input, and reduces the odds of making a mistake. Once we know `getValueFromUser` works, we can call it as many times as we desire.

This is the essence of modular programming: the ability to write a function, test it, ensure that it works, and then know that we can reuse it as many times as we want and it will continue to work (so long as we don't modify the function -- at which point we'll have to retest it).

Best practice

Follow DRY: "don't repeat yourself". If you need to do something more than once, consider how to modify your code to remove as much redundancy as possible. Variables can be used to store the results of calculations that need to be used more than once (so we don't have to repeat the calculation). Functions can be used to define a sequence of statements we

want to execute more than once. And loops (which we'll cover in a later chapter) can be used to execute a statement more than once.

Like all best practices, DRY is meant to be a guideline, not an absolute. Reader Yariv has noted (https://www.learncpp.com/cpp-tutorial/function-return-values-value-returning-functions/#comment-593257)[5] that DRY can harm overall comprehension when code is broken into pieces that are too small.

As an aside...

The opposite of DRY is WET ("Write everything twice").

Conclusion

Return values provide a way for functions to return a single value back to the function's caller.

Functions provide a way to minimize redundancy in our programs.

Quiz time

Question #1

Inspect (do not compile) each of the following programs. Determine what the program will output, or whether the program will generate a compiler error.

Assume you have "treat warnings as errors" turned off.

1a)

```
#include <iostream>

int return7()
{
    return 7;
}

int return9()
{
    return 9;
}

int main()
{
    std::cout << return7() + return9() << '\n';

    return 0;
}
```

Show Solution (javascript:void(0))[6]

1b)

```
#include <iostream>

int return7()
{
    return 7;

    int return9()
    {
        return 9;
    }
}

int main()
{
    std::cout << return7() + return9() << '\n';

    return 0;
}
```

Show Solution (javascript:void(0))[6]

1c)

```
#include <iostream>

int return7()
{
    return 7;
}

int return9()
{
    return 9;
}

int main()
{
    return7();
    return9();

    return 0;
}
```

Show Solution (javascript:void(0))[6]

1d)

```
#include <iostream>

int getNumbers()
{
    return 5;
    return 7;
}

int main()
{
    std::cout << getNumbers() << '\n';
    std::cout << getNumbers() << '\n';

    return 0;
}
```

Show Solution (javascript:void(0))[6]

1e)

```
#include <iostream>

int return 5()
{
    return 5;
}

int main()
{
    std::cout << return 5() << '\n';

    return 0;
}
```

Show Solution (javascript:void(0))[6]

1f) Extra credit: Will the following program compile?

```
#include <iostream>

int returnFive()
{
    return 5;
}

int main()
{
    std::cout << returnFive << '\n';

    return 0;
}
```

Show Solution (javascript:void(0))[6]

Question #2

What does “DRY” stand for, and why is it a useful practice to follow?

Show Solution (javascript:void(0))[6]

Next lesson

2.3 Void functions (non-value returning functions)

2

Back to table of contents

8

Previous lesson

2.1 Introduction to functions

9

10

B U URL INLINE CODE C++ CODE BLOCK HELP!

Leave a comment...

Name*

Email*

Notify me about replies:

POST COMMENT

Find a mistake? Leave a comment above!

Avatars from https://gravatar.com/[13] are connected to your provided email address.

399 COMMENTS Newest

Likss

April 9, 2024 12:51 am

Hey , If i have to make a function that calculates area of an rectangle. I do not think i can use the same example as above since i have to first show " enter length" and then " enter

breadth". Is there any way to lessen the length of the code . Kindly suggest if there is any way (context : do not suggest any method i have not learned yet i.e . Till lecture 2.3

0 Reply

Rez

Reply to Likss [14] April 9, 2024 11:19 am

You can pass the entire message or a part of it as input to the function.

```
#include <iostream>
#include <string>

// Function to get an integer value from the user with a prompt
int getValueFor(std::string prompt) {
    std::cout << "Enter the " << prompt << ": ";
    int input{};
    std::cin >> input;
    return input;
}

// Function to calculate the area of a rectangle
int calculateArea() {
    int length = getValueFor("length");
    int breadth = getValueFor("breadth");
    return length * breadth;
}

int main() {
    std::cout << "The area of the rectangle is: " <<
calculateArea() << std::endl;
    return 0;
}
```

Last edited 8 days ago by Rez

0 Reply

Alex Author

Reply to Rez [15] April 10, 2024 4:44 pm

Use `std::string_view` instead of `std::string` for your function parameter to avoid making an unnecessary copy of the argument string.

1 Reply

Linus

April 5, 2024 7:02 pm

```cpp
#include <iostream>

int Calculate(int numA, int numB);
int GetValueFromUser();
int main()
{
    int numA{GetValueFromUser()};
    int numB{GetValueFromUser()};
    Calculate(numA, numB);
}
int Calculate(int numA, int numB) {
    std::cout << numA << " + " << numB << " = " << numA + numB;
    return EXIT_SUCCESS;
}

int GetValueFromUser() {
    std::cout << "Type int ";
    int input;
    std::cin >> input;
    return input;
}
```

0 Reply

Rexxy

March 5, 2024 1:34 pm

For the first part i like to try my hand at the goal first then see what the lesson says, and i noticed i could do the first section like this (sorry if the code looks a mess i comment out the solution im not using but leave both in there) could you please tell me the difference between the two methods

```cpp
#include <iostream>
//by making the vairible out of either function
//i seem to be able to call in in multiple functions
int num{};


     void getValueFromUserAttempt1()
 {
     std::cout << "Please enter an interger: ";

     std::cin >> num;
 }

/* int getValueFromUserSolution()
{
    std::cout << "Please enter an interger: ";
        int imput{};
        std::cin >> imput;

        return imput;
}
*/


int main()
{
    //int num{ getValueFromUserSolution() };
    getValueFromUserAttempt1();
    std::cout << num << " doubled is: " << num * 2;

    return 0;
}
```

0 Reply

Allex

Reply to Rexxy [16] March 11, 2024 3:09 am

Creating global variables is not a good approach

2 Reply

Rexxy

Reply to Rexxy [16] March 5, 2024 1:43 pm

Nevermind after a bit more testing and looking at the code where my way could only be used to change the value of int num the prefered way can be used whenever i need a new number so like if i want to multiply two numbers i can make num1 and num2 both run the same code where as before i wouldnt be able to

0 Reply

gagoogla
February 27, 2024 8:24 am

```
#include  <iostream>

int getval()
{
   std::cout << "Pls type integer" ;

    int x {} ;
    std::cin >> x ;


    return x ;  //x is now returned back

}



int  hiMultiply ( int a)
 {



     return a*3*4 ;

 }


  int hiDiv (int a )
  {



     return a/2 ;


  }


int hiSub(int a )
{

   return a-2 ;

}
int hiAdd(int a )
{


   return a+a ;

}

int hiSquared(int a)
{


   return a*a ;

}


/// @brief
/// @return
int main ()
{
```

```
      std::cout << "starting main " << '\n' ;
    int y { getval ()} ; // now x is returned and stored at y variable so yeah x =y now after do add is executed we see that paused program is now restarted


    std::cout << "Here is your number" << y ;




       std::cout << "do you need a double of program perhaps a multiplication of it division lemme give out all of the stuff" <<'\n' ;

     hiMultiply(y);
     hiSub ( y ) ;
     hiAdd(y );
     hiDiv( y  );
     hiSquared( y ) ;

    return 0 ;
}
```

0 Reply

gagoogla

Reply to gagoogla [17] February 27, 2024 8:31 am

tf my code not working i made another program with fucking loops it works

```
#include  <iostream>

int main ()
{
  char operatprs;
  float num1, num2;

  std::cout << "Enter operator: +, -, *, /: ";  //chooses an
operator
  std::cin >> operatprs;  //operator printed

 std::cout << "Enter two operands: ";
  std::cin >> num1 >> num2;  //i hate namespaces be it a local or
global scope

  switch(operatprs) {

    case '+':
      std::cout << num1 << " + " << num2 << " = " << num1 + num2;
      break;

    case '-':
      std::cout << num1 << " - " << num2 << " = " << num1 - num2;
      break;

    case '*':
      std::cout << num1 << " * " << num2 << " = " << num1 * num2;
      break;

    case '/':
      std::cout << num1 << " / " << num2 << " = " << num1 / num2;
      break;

    default:
      // If the operator is other than +, -, * or /, error message
is shown
      std::cout << "oi mate enter correct stuff or i will empty
your pantaloon compartments and ask you your bugati color imma be
real here";
      break;
}
 return 0 ;
}
```

yes i wrote this code and it fucking works i try to call functions inside functions this shit does not work pls help me

0 Reply

gagoogla

Reply to gagoogla [18] February 27, 2024 8:40 am

pls help me

0 Reply

gagoogla

Reply to gagoogla [19] February 27, 2024 8:38 pm

nvm i made a new calculator with enhanced functionality

```cpp
#include <iostream>

int getval() {
  std::cout << "Please type an integer: ";

  int x;
  std::cin >> x;

  return x; // Return the input value
}

int multiply(int a) {
  return a * 3 * 4; // Perform multiplication
}

int divide(int a) {
  return a / 2; // Perform division
}

int subtract(int a) {
  return a - 2; // Perform subtraction
}

int add(int a) {
  return a + a; // Perform addition
}

int square(int a) {
  return a * a; // Perform squaring
}

int main() {
  std::cout << "Starting main\n";

  int y = getval(); // Get the user's input and store it in y

  std::cout << "Here is your number: " << y << std::endl;

  std::cout << "\nDo you need any calculations? (Choose an option by number)\n"
            << "1. Multiplication (double)\n"
            << "2. Division (half)\n"
            << "3. Addition (double)\n"
            << "4. Subtraction (minus 2)\n"
            << "5. Square\n"
            << "Enter your choice: ";

  int choice;
  std::cin >> choice;

  switch (choice) {
    case 1:
      std::cout << y << " * 3 * 4 = " << multiply(y) << std::endl;
      break;
    case 2:
      std::cout << y << " / 2 = " << divide(y) << std::endl;
      break;
    case 3:
      std::cout << y << " + " << y << " = " << add(y) << std::endl;
      break;
    case 4:
```

```
    case 4:
      std::cout << y << " - 2 = " << subtract(y) <<
std::endl;
      break;
    case 5:
      std::cout << y << " squared = " << square(y) <<
std::endl;
      break;
    default:
      std::cout << "Invalid choice." << std::endl;
  }

  return 0;
}

 //problem was that i did not force the printing of
functions on to the terminal and yeah i used endl i know
it is a bad practice wrote this code in hurry imma keep
on adding stuff to this like scientific compute and
error handling a bit better
```

1 Reply

Augustas
February 20, 2024 11:38 pm

These lessons have great explanations and questions but for best understanding actual coding should be done. Do you know were to find good exercises for all levels?

0 Reply

IceFloe
February 19, 2024 2:40 am

1)And if the type of the return value and the type of the function are different, what will be the result in the end?

```
int value()
{
 return 2.5
}
```

2. the second equally important question is, does each function call occupy a separate memory location? I mean, writing modular code, as you said, may be practical, but from the point of view of optimization it is unprofitable.
3. it turns out that of all the functions, only void does not return have a value and only it can be used without the `return` operator?
4. What should I do if I want to return more than two values from one function?

5)Thanks for the lesson, once again I apologize for asking questions too often, no one clarifies such nuances on the Internet except you

Last edited 1 month ago by IceFloe

2 Reply

Alex Author

Reply to IceFloe [20] February 19, 2024 8:40 pm

1. This performs a conversion from the type of the return value to the function's return type. We talk a lot more about conversions in a future chapter.
2. Generally speaking, each function is compiled into a single bit of machine code that can be reused with each function call. Each time the function is actually called, some memory must be set aside to create the function's parameters and local variables. This memory is then returned when the function returns back to the caller (as those variablers are no longer needed).
3. Yes
4. You have a variety of options to return multiple values. The most easily understood is to create a struct (a custom type that contains multiple values) and return an instance of that struct. Other options include returning a std::vector, using out-parameters, or using structured binding. Most of these are covered in future chapters.

3 Reply

Qwerty

February 17, 2024 4:21 pm

```
#include <iostream>

int doesThisWork();

int main()
{
    std::cout << "practice \n \n";
    int num{ doesThisWork() };
    int num2{ doesThisWork() };
    std::cout << "the two multiplied together is: \n" << num * num2 << '\n';

    return 0;
}

int doesThisWork()
{
    std::cout << "Enter an Integer: ";
    int num{};
    std::cin >> num;
    std::cout << '\n';

    return num;
}
```

Experimenting is fun!

0 Reply

Yariv

February 5, 2024 8:35 am

The concept of DRY is not absolute. Do NOT force yourself to never repeat code as it could lead to much less understandable and much harder to maintain code.

A common situation is where you have two behaviors that are ALMOST identical, but the subtle differences between them means you have to try to break them up into smaller bits so you can reuse the identical parts. Now imagine a few days later you need a third behavior that's again slightly different. Now you have to either break apart more and more of the common functionality or provide "context" to the functions to modify their behavior at runtime. Its very easy to end up with an uncomprehensible pasta of really small bits of code that by themselves barely have any meaning left, or with huge functions that have so much "context" required to work with its no longer clear what they even do or how they should/can be used.

Do yourself a favor and always take best practices with a grain of salt. In my opinion 99% of the time you should prioritize readable and maintainable code over sticking to best practices. Knowing when to stick to a best practice and when not to comes with experience - just know that the option is there :)

22 Reply

Alex Author

Reply to Yariv [5] February 6, 2024 12:38 pm

Great comment on both DRY and a reminder that best practices are guidelines, not absolutes! Best practices exist because most of the time they lead to the desired result of readable and maintainable code. But if they are ever in conflict, whatever produces the most readable and maintainable code wins.

17 Reply

Jordan

January 31, 2024 11:12 pm

```cpp
#include <iostream>

using namespace std;

//funtion for header for practice function using integer.
int practiceFunc(){

  std::cout << " Enter an integer value: ";

  int Num{};   //variable declaration
  std::cin >> Num; //get the value from the user
  std::cout <<'\n';

  return Num; //return the value to the caller meaning the int main().
}

int main(){

  std::cout << " This is a practice function using integer \n";
  std::cout <<'\n';
  int num = practiceFunc(); //initialized num to practiceFunc(). this is the same as int num{practiceFunc()};

  std::cout << "The triple of the value entered is: ";
  std::cout << num * 3 << '\n'; //called the value from the int function

return 0;
}
```

it's my first time actually learning what a function is and how to actually define a function in a program. I really am grateful for the awesome work you are doing for us. For real thank you.

0 Reply

Aaron

January 26, 2024 9:58 pm

When I try to step into or run to cursor in this code to see where it goes wrong (for practice) I receive the message "There are build errors. Would you like to continue and run the last successful build?"

I thought the point of debugging is to see where code goes wrong? Why can't I debug the following code?

#include <iostream>

void getValueFromuser()
{
std::cout << "Enter an integer: ";
int input{};
std::cin >> input;
}

```
int main()
{
getValueFromUser();

int num{};

std:: cout << num << " doubled is: \n" << num * 2 << '\n';

return 0;
}
```

Last edited 2 months ago by Aaron

0 Reply

Alex Author

Reply to Aaron [21] January 28, 2024 4:46 pm

Debugging is for finding errors at runtime. If your code isn't building, then the issue is occurring at compile-time. Your compiler should be telling you (via a compilation error) what the issue is.

In this case, you are calling `getValueFromUser()` but you've named the function `getValueFromuser()`. C++ is case sensitive.

1 Reply

Mridul Thakur

Reply to Alex [22] February 5, 2024 7:24 pm

lol, i wonder if he is still stuck into this problem. I remember back when i started learning programming. I searched whole internet for a problem, in the end it was just a typo.

0 Reply

Aaron

January 22, 2024 6:52 am

Can someone recommend a resource for more quiz questions? I worked my way to the end of section 3 and realized I need to go through each section again to REALLY get this. However, it will help if I can have more questions to solve. I tried to find good videos/tutorials outside this and was very dismayed. This is the best so far by far.

0 Reply

mechta

Reply to Aaron [23] January 22, 2024 12:03 pm

> Try out sites like codeabbey and hackerrank (some stuff will be too hard in the beginning, but you will gain the principles as you go through this site - it's helping me thus far).
>
> 0 Reply

james
January 14, 2024 12:42 pm

amazing ++

0 Reply

Peter
January 14, 2024 10:34 am

These lessons are really starting to build my confidence in learning C++!

2 Reply

edge
January 8, 2024 7:39 am

Thanks♪(･ω･)ﾉ

0 Reply

Abayomi
January 3, 2024 1:21 pm

I will never look functions the same way again, anytime I learn a new programming when I get to Function it usually such an headache for me. Thank you for the simplity..

1 Reply

Aaron
December 16, 2023 12:38 am

Greetings. Thak you for this tutorial. I am a complete beginner and this makes c++ approachable. Please help me make this adjustment to the code in this module. I will post my code, then the output, then what I want the desired output to be.

```
#include <iostream>

int userInput()
{
```

```
std::cout << "Enter an integer: ";
int user{};
std::cin >> user;

return 55; // return the value the user entered back to the caller
}

int main()
{
int num { userInput() }; // initialize num with the return value of getValueFromUser()

std::cout << num << " tripled is: " << num * 3 << '\n';

return 0;
```

RETURN:

"Enter an integer: 10
55 tripled is: 165"

WHAT I WANT:

Instead of "55 tripled...,"

I want it to say "10 tripled is: 165"

This does not make logical or arithmetic sense, however, I just want to know how to change how numbers "look" while maintaining their "real" value.

Thank you in advance for your suggestions.

0 Reply

Anon

Reply to Aaron [24] January 21, 2024 10:06 pm

Here's how I fixed it:

```
#include <iostream>

int userInput()
    {
        std::cout << "Enter an integer: ";
        int user{};
        std::cin >> user;

        return user; // return the value the user entered back to
the caller
    }

int main()
    {
        int num { userInput() }; // initialize num with the return
value of getValueFromUser()
        std::cout << num << " tripled is: " << num * 3 << '\n';

        return 0;
    }
```

I just changed `return 55` to `return user` in line 9.

Last edited 2 months ago by Anon

0 Reply

Alex Author

Reply to Aaron [24] December 18, 2023 2:59 pm

`userInput()` should return `num`, then the print statement in `main()` should use `55 * 3` instead of `num * 3`.

1 Reply

AbeerOrTwo

December 14, 2023 8:42 pm

This was a good lesson to end for the night, I work more tomorrow

1 Reply

Krishnakumar

November 15, 2023 5:02 am

> it's a good idea to document your function with a comment indicating what the return values mean.

I feel like this is an opportunity to introduce Doxygen comments to generate project documentation.

0 Reply

Krishnakumar
November 15, 2023 4:56 am

>C does allow main() to be called explicitly, so some C++ compilers will allow this for compatibility reasons.

Will these compilers allow this even flags requesting strict compliance behaviour such as `-Wpedantic` and `-pedantic-error` flags are used?

Last edited 5 months ago by Krishnakumar

0 Reply

Alex Author
Reply to Krishnakumar [25] November 15, 2023 12:52 pm

Yes, Clang and GCC both flag calling `main()` explicitly as an error in pedantic mode.

0 Reply

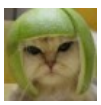
Rich
November 13, 2023 8:39 pm

Hello i looked through some of the replies and didnt see my question but didnt go through all so i apologize if it was answered.

In 1f) Extra Credit when you state the program will compile, but the function will not be called because of the missing (). I expected it not to compile at all because of the missing () or " ". But when i wrote the program myself and ran it i was surprised to see it ran and printed out 00007FF674D814D3. I am assuming this is because int returnFive() ALSO created a variable of returnFive not just a function?

Last edited 5 months ago by Rich

1 Reply



Alex Author
Reply to Rich [26] November 14, 2023 9:46 pm

No, it's because `returnFive` without the parenthesis implicitly converts to a function pointer, and on some compilers printing a function pointer prints the address where the function resides in memory.

0 Reply

Rich

Reply to Alex [27] November 14, 2023 9:52 pm

ok thanks! don't think i've gotten to function pointers yet! But i was curious what was going on. Thanks again and great tutorial you have put together. I am learning a ton!

0 Reply

Links

1. https://www.learncpp.com/author/Alex/
2. https://www.learncpp.com/cpp-tutorial/void-functions-non-value-returning-functions/
3. https://www.learncpp.com/cpp-tutorial/introduction-to-the-preprocessor/
4. https://www.learncpp.com/cpp-tutorial/uninitialized-variables-and-undefined-behavior/
5. https://www.learncpp.com/cpp-tutorial/function-return-values-value-returning-functions/#comment-593257
6. javascript:void(0)
7. https://www.learncpp.com/cpp-tutorial/keywords-and-naming-identifiers/#rules
8. https://www.learncpp.com/
9. https://www.learncpp.com/cpp-tutorial/introduction-to-functions/
10. https://www.learncpp.com/function-return-values-value-returning-functions/
11. https://www.learncpp.com/cpp-tutorial/developing-your-first-program/
12. https://www.learncpp.com/cpp-tutorial/chapter-2-summary-and-quiz/
13. https://gravatar.com/
14. https://www.learncpp.com/cpp-tutorial/function-return-values-value-returning-functions/#comment-595575
15. https://www.learncpp.com/cpp-tutorial/function-return-values-value-returning-functions/#comment-595600
16. https://www.learncpp.com/cpp-tutorial/function-return-values-value-returning-functions/#comment-594320
17. https://www.learncpp.com/cpp-tutorial/function-return-values-value-returning-functions/#comment-594082
18. https://www.learncpp.com/cpp-tutorial/function-return-values-value-returning-functions/#comment-594084
19. https://www.learncpp.com/cpp-tutorial/function-return-values-value-returning-functions/#comment-594085
20. https://www.learncpp.com/cpp-tutorial/function-return-values-value-returning-functions/#comment-593810
21. https://www.learncpp.com/cpp-tutorial/function-return-values-value-returning-functions/#comment-592937

22. https://www.learncpp.com/cpp-tutorial/function-return-values-value-returning-functions/#comment-592997
23. https://www.learncpp.com/cpp-tutorial/function-return-values-value-returning-functions/#comment-592682
24. https://www.learncpp.com/cpp-tutorial/function-return-values-value-returning-functions/#comment-591081
25. https://www.learncpp.com/cpp-tutorial/function-return-values-value-returning-functions/#comment-589825
26. https://www.learncpp.com/cpp-tutorial/function-return-values-value-returning-functions/#comment-589740
27. https://www.learncpp.com/cpp-tutorial/function-return-values-value-returning-functions/#comment-589807